

Jerarquía de Memorias

• Conceptos Avanzados de Memoria Cache

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya





Índice

Introducción

- Cualquier optimización a realizar tiene como objetivo final reducir el tiempo de ejecución:

$$T_{exe} = N \cdot CPI \cdot T_c$$

$$T_{exe} = N \cdot (CPI_{id} + CPI_{mem}) \cdot T_c$$

$$T_{exe} = N \cdot (CPI_{id} + n_r \cdot m \cdot t_{pf} + CPI_{WR}) \cdot T_c$$

- Hay elementos que dependen del lenguaje máquina (N, n_r) en los cuales no influye la jerarquía de memoria.
- La jerarquía de memoria puede influir en:
  - Tasa de fallos (objetivo $m\downarrow$)
  - Tiempo de penalización por fallo (objetivo $t_{pf}\downarrow$)
  - Coste de las escrituras (objetivo $CPI_{WR}\downarrow$)
  - Ancho de banda con memoria



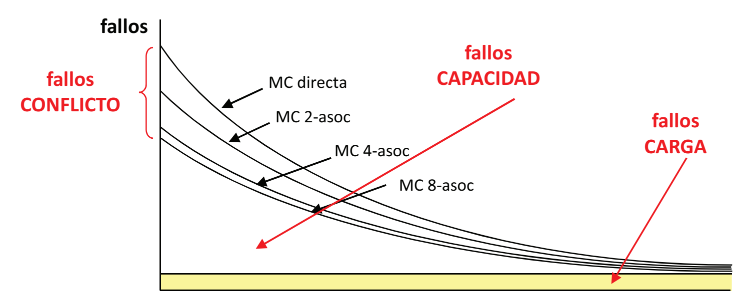
Introducción

Los fallos de cache pueden dividirse en tres categorías:

- Carga (compulsory): se producen la primera vez que se accede a una posición de memoria.
- Capacidad: todas las líneas que necesita un programa no caben en la Memoria Cache.
- Conflicto: se producen cuando varias líneas se mapean en el mismo lugar de la MC (sólo en MC directas y asociativas por conjuntos)
- [En los multiprocesadores aparecen los fallos de Coherencia]

¿Cómo se calculan?: ¡Ayuda a entenderlo!

Entenderlos: ¡Ayuda a reducirlos!



Introducción

Técnicas básicas (obvias) para mejorar el rendimiento de la cache:

- **Aumentar el tamaño de bloque (m↓)**. Reduce los fallos de carga, pero puede ser contraproducente (**m↑**).
- **Aumentar el tamaño de cache (m↓).** Reduce los fallos de capacidad (y conflicto), pero aumenta el tiempo de acceso a la cache (**tsa↑**) y el consumo (**W↑**).
- **Aumentar el grado de asociatividad (m↓)**. Reduce los fallos de conflicto, pero aumenta el tiempo de acceso a la cache (**tsa↑**).
- **Caches multinivel (t_{pf}↓).** L1 pequeña (**m↑ y tsa↓**) y L2 grande (**m↓ y tsa↑**).
- **Dar más prioridad a las lecturas que a las escrituras (CPI_{WR}↓).** El coste de las escrituras se puede reducir (ocultar) utilizando buffers de escrituras.



Introducción

Las Técnicas avanzadas para mejorar el rendimiento de la cache se pueden agrupar en:

- Reducir el coste de un acierto en cache (**tsa ↓**): caches pequeñas y simples, predicción de vía y trace caches.
- Aumentar el ancho de banda de cache(**BW↑**): caches segmentadas, caches multibanco y caches no bloqueantes.
- Reducir el coste de los fallos(t_{pf}↓): early restart y merging write buffers.
- Reducir la tasa de fallos (**m↓**): optimizaciones del compilador.
- Reducir el coste de los fallos(t_{pf}↓) y la tasa de fallos (**m↓**) vía paralelismo: prebúsqueda hardware y prebúsqueda software.

Optimizaciones

1) Caches pequeñas y simples para reducir el tiempo de acceso en acierto

2) Predicción de vía para reducir el tiempo de acceso en acierto

3) Trace Caches para reducir el tiempo de acceso en acierto

4) Caches segmentadas para aumentar el ancho de banda de la cache

5) Non Blocking caches

6) Caches multibanco

7) Reducir la penalización por fallo: early restart, transferencia en desorden

8) Buffers de escritura

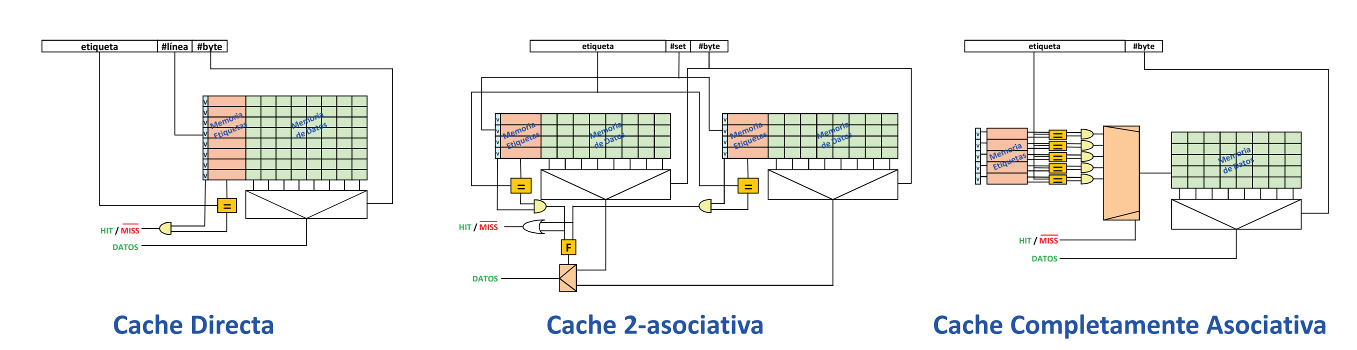
9) Optimizaciones de código para reducir tasa fallos

10) Prefetch de instrucciones y/datos para reducir tasa de fallos y/o la penalización por fallo



Caches Simples para reducir el tiempo de acceso

- **El tiempo de acceso a la cache depende del camino crítico**

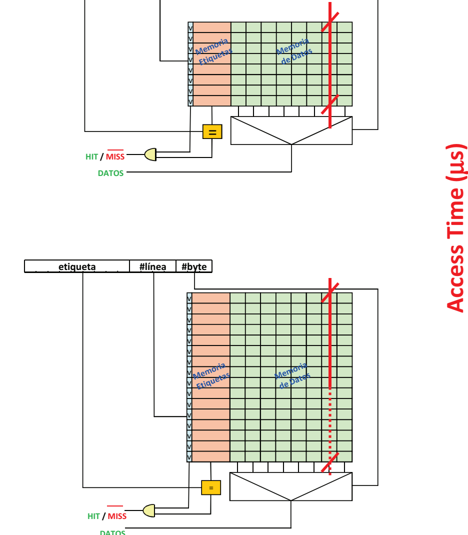


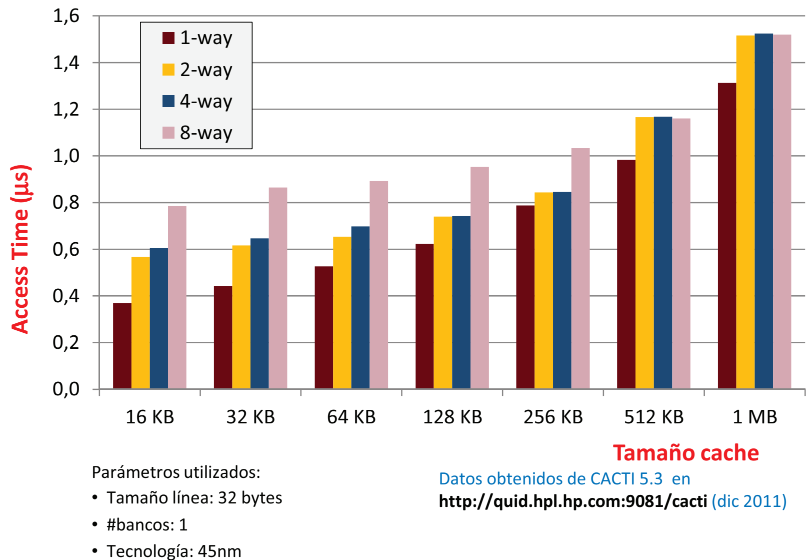
Cache Directa   Cache 2-asociativa   Cache Completamente Asociativa

- **Cache Directa: Acceso (Memoria datos y etiquetas), comparar etiqueta y validar línea.**
- **Cache Asociativa por conjuntos: Acceso (Memoria datos y etiquetas), comparar etiqueta, validar línea y seleccionar vía.**
- **Cache Completamente Asociativa: Acceso (Memoria etiquetas), comparar etiqueta, validar línea y Acceso (Memoria datos).**
- **El tiempo de acceso a una cache directa es menor que el de una cache asociativa capacidad equivalente.**

Caches Pequeñas para reducir el tiempo de acceso

- El tiempo de acceso a la cache depende del tamaño de la cache (a mayor capacidad aumenta el tamaño del array de memoria, de los decodificadores, …)





Parámetros utilizados:
- Tamaño línea: 32 bytes
- #bancos: 1
- Tecnología: 45nm

Datos obtenidos de CACTI 5.3 en **http://quid.hpl.hp.com:9081/cacti** (dic 2011)



Caches Pequeñas para reducir el tiempo de acceso

- La mayoría de las caches de primer nivel de los procesadores actuales son pequeñas :

Procesador	L1 Datos	L1 Instrucciones	L2 unificada	L3 unificada
Intel Xeon E5502	32 KB	32 KB	256 KB	4096 KB
Intel Core i3-560	32 KB	32 KB	256 KB	4096 KB
Intel Core i5-680	32 KB	32 KB	256 KB	4096 KB
Intel Core i7-880	32 KB	32 KB	256 KB	8192 KB
AMD Athlon II X2 255	64 KB	64 KB	1024 KB	-
AMD Opteron 8382	64 KB	64 KB	512 KB	6144 KB

Enero-2011

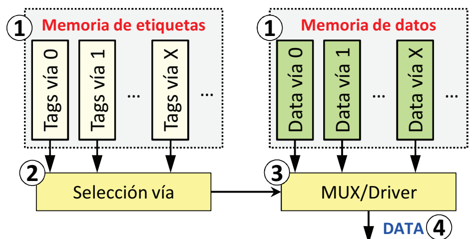
Optimizaciones

1) Caches pequeñas y simples para reducir el tiempo de acceso en acierto
2) **Predicción de vía para reducir el tiempo de acceso en acierto**
3) Trace Caches para reducir el tiempo de acceso en acierto
4) Caches segmentadas para aumentar el ancho de banda de la cache
5) Non Blocking caches
6) Caches multibanco
7) Reducir la penalización por fallo: early restart, transferencia en desorden
8) Buffers de escritura
9) Optimizaciones de código para reducir tasa fallos
10) Prefetch de instrucciones y/datos para reducir tasa de fallos y/o la penalización por fallo

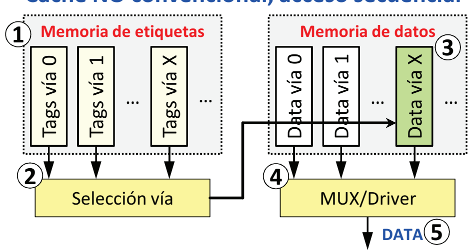


Predicción de vía para reducir el tiempo de acceso

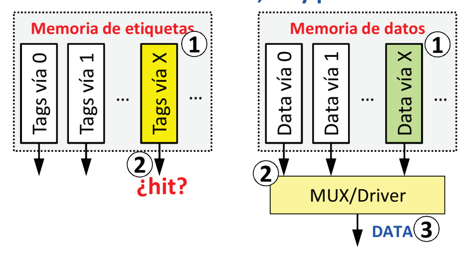
Cache convencional, acceso paralelo



Cache NO convencional, acceso secuencial



Cache NO convencional, way prediction

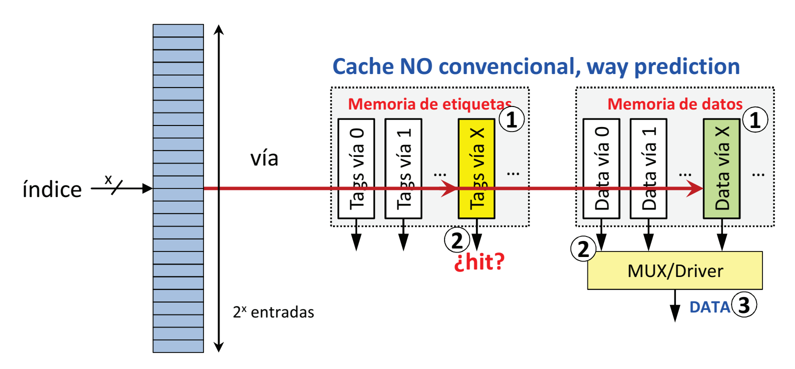


- En una cache convencional, con acceso en paralelo, el consumo es elevado. El tiempo de acceso viene determinado por el acceso a memoria y la selección de la vía.
- Una cache con acceso secuencial es lenta, pero reduce sustancialmente el consumo porque sólo accede a los datos de la vía seleccionada.
- En una cache con way prediction, el consumo se reduce porque sólo accedemos a la vía indicada por la predicción y reduce el tiempo de acceso en caso de acierto, porque no necesita la selección de vía.

Esquemas obtenidos en Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi and Kaushik Roy. «Reducing set-associative cache energy via way-prediction and selective direct-mapping», Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (MICRO 34), 2001.

Predicción de vía para reducir el tiempo de acceso

- **¿Cómo se realiza la predicción?**



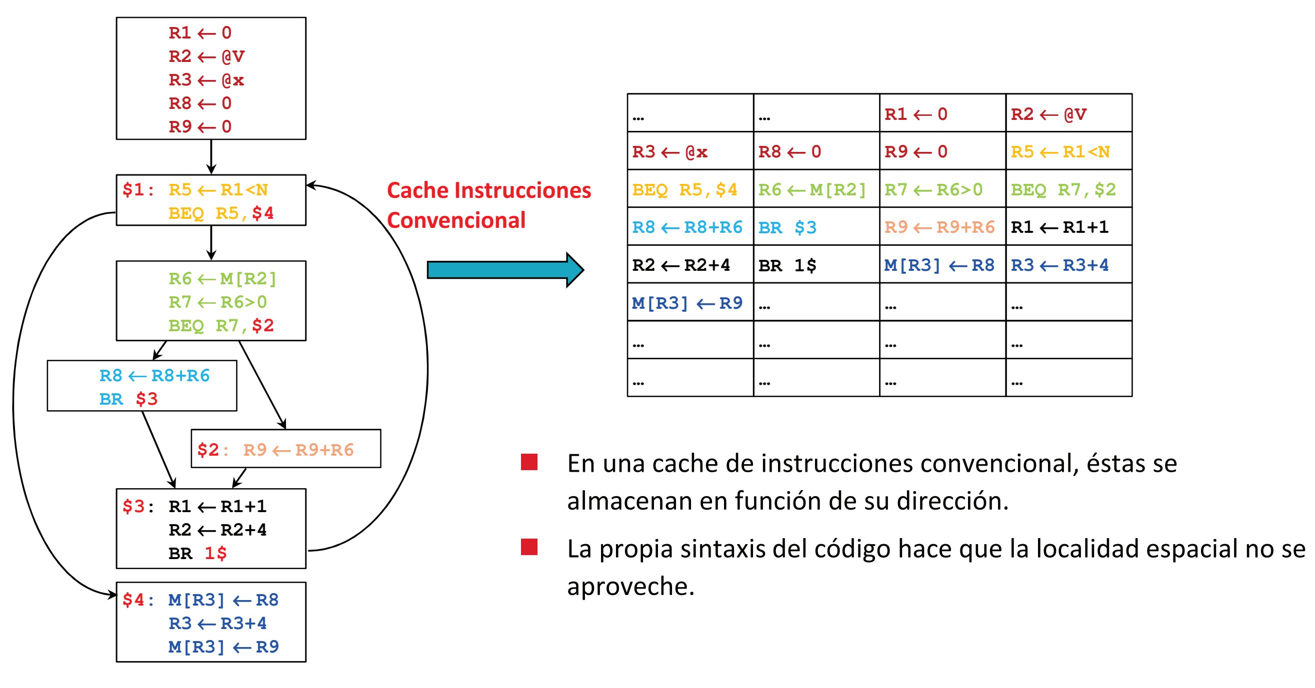
- **El índice podría ser el PC de la instrucción en ejecución ⇒ La tabla sería demasiado grande.**
- **Podemos utilizar unos pocos bits del PC (como en una cache).**
- **La tasa de aciertos en la predicción depende (entre otras cosas) del tamaño de la tabla (2^x)**
- **Los programas tienen una «ejecución predecible» (localidad).**
- **La tabla de predicción se actualiza con el comportamiento de los accesos previos.**



Optimizaciones

1) Caches pequeñas y simples para reducir el tiempo de acceso en acierto
2) Predicción de vía para reducir el tiempo de acceso en acierto
3) **Trace Caches para reducir el tiempo de acceso en acierto**
4) Caches segmentadas para aumentar el ancho de banda de la cache
5) Non Blocking caches
6) Caches multibanco
7) Reducir la penalización por fallo: early restart, transferencia en desorden
8) Buffers de escritura
9) Optimizaciones de código para reducir tasa fallos
10) Prefetch de instrucciones y/datos para reducir tasa de fallos y/o la penalización por fallo

Trace Cache

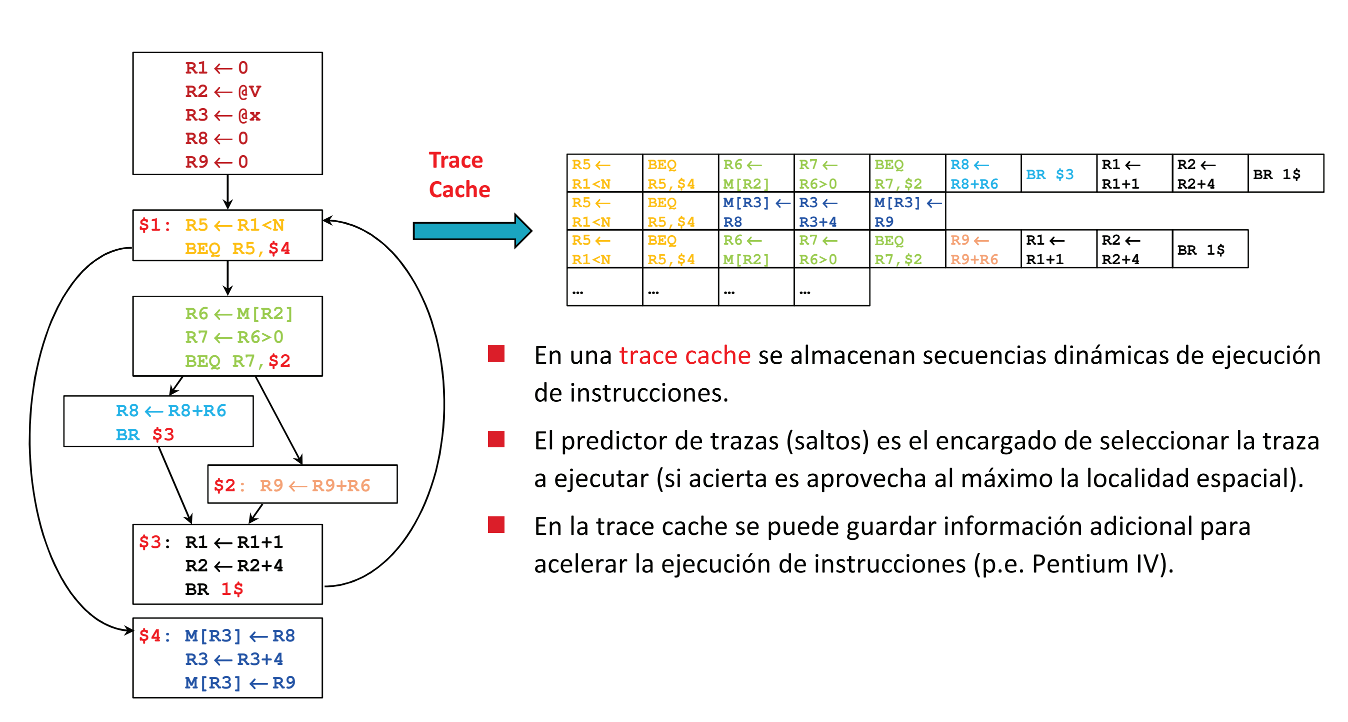


...	...	R1 ← 0	R2 ← @V
R3 ← @x	R8 ← 0	R9 ← 0	R5 ← R1<N
BEQ R5,$4	R6 ← M[R2]	R7 ← R6>0	BEQ R7,$2
R8 ← R8+R6	BR $3	R9 ← R9+R6	R1 ← R1+1
R2 ← R2+4	BR 1$	M[R3] ← R8	R3 ← R3+4
M[R3] ← R9	...	...	...
...	...	...	...
...	...	...	...

- En una cache de instrucciones convencional, éstas se almacenan en función de su dirección.
- La propia sintaxis del código hace que la localidad espacial no se aproveche.



Trace Cache



R5 ← R1<N	BEQ R5,$4	R6 ← M[R2]	R7 ← R6>0	BEQ R7,$2	R8 ← R8+R6	BR $3	R1 ← R1+1	R2 ← R2+4	BR 1$
R5 ← R1<N	BEQ R5,$4	M[R3] ← R8	R3 ← R3+4	M[R3] ← R9					
R5 ← R1<N	BEQ R5,$4	R6 ← M[R2]	R7 ← R6>0	BEQ R7,$2	R9 ← R9+R6	R1 ← R1+1	R2 ← R2+4	BR 1$	
...	...	...	...						

- En una trace cache se almacenan secuencias dinámicas de ejecución de instrucciones.
- El predictor de trazas (saltos) es el encargado de seleccionar la traza a ejecutar (si acierta es aprovecha al máximo la localidad espacial).
- En la trace cache se puede guardar información adicional para acelerar la ejecución de instrucciones (p.e. Pentium IV).

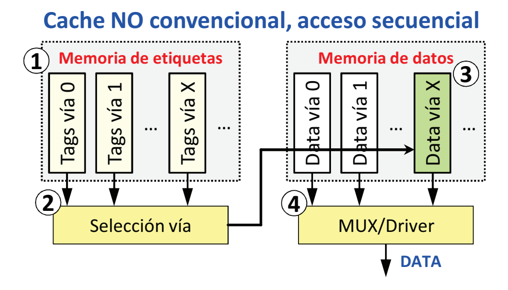
Optimizaciones

1) Caches pequeñas y simples para reducir el tiempo de acceso en acierto
2) Predicción de vía para reducir el tiempo de acceso en acierto
3) Trace Caches para reducir el tiempo de acceso en acierto
4) **Caches segmentadas para aumentar el ancho de banda de la cache**
5) Non Blocking caches
6) Caches multibanco
7) Reducir la penalización por fallo: early restart, transferencia en desorden
8) Buffers de escritura
9) Optimizaciones de código para reducir tasa fallos
10) Prefetch de instrucciones y/datos para reducir tasa de fallos y/o la penalización por fallo

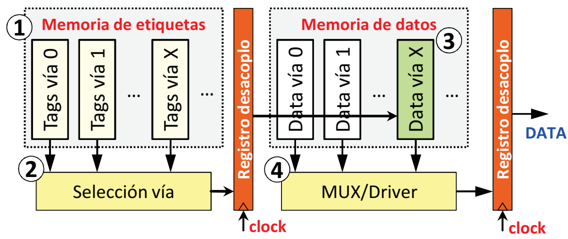


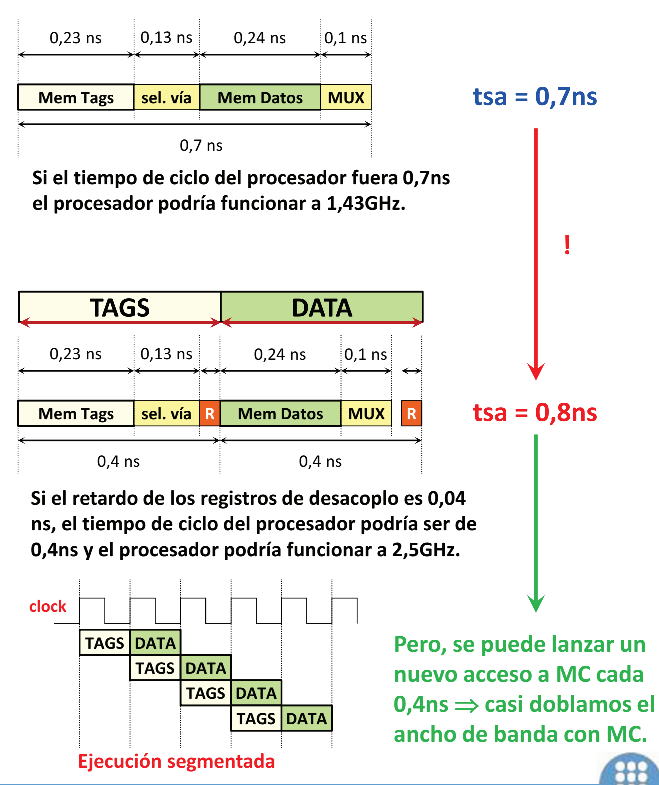
Cache segmentada

Cache NO convencional, acceso secuencial

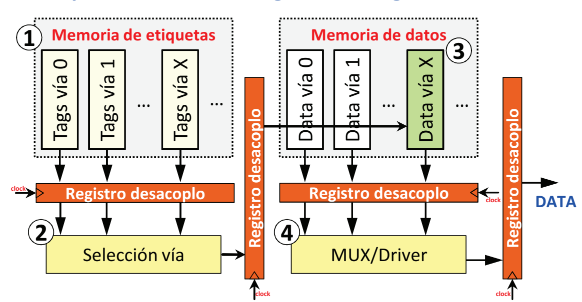


Cache NO convencional, segmentada

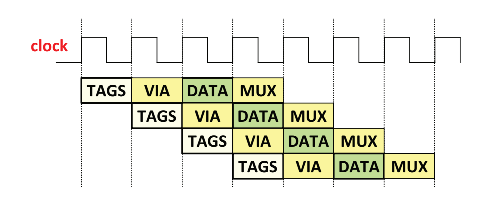




Cache segmentada



tsa = 1,12ns



clock

TAGS	VIA	DATA	MUX			
	TAGS	VIA	DATA	MUX		
		TAGS	VIA	DATA	MUX	
			TAGS	VIA	DATA	MUX

MC (accesos de 4B)	tsa	Tc	f	Ancho banda máx.
NO segmentada	0,7ns	0,7ns	1,43GHz	5,71 GB/s
Segmentada 2 etapas	0,8ns	0,4ns	2,5GHz	10 GB/s ≈x2
Segmentada 4 etapas	1,12ns	0,28ns	3,57GHz	14,29 GB/s ≈x3

La latencia de 1 acceso individual aumenta (tsa=1,12ns), pero el ancho de banda aumenta porque se pueden realizar 4 accesos en paralelo (se puede iniciar un acceso a MC cada 0,28ns).

La segmentación, junto con los conceptos de cache y paralelismo, es uno de los paradigmas básicos en el diseño de procesadores.



Optimizaciones

1) Caches pequeñas y simples para reducir el tiempo de acceso en acierto
2) Predicción de vía para reducir el tiempo de acceso en acierto
3) Trace Caches para reducir el tiempo de acceso en acierto
4) Caches segmentadas para aumentar el ancho de banda de la cache
5) **Non Blocking caches**
6) Caches multibanco
7) Reducir la penalización por fallo: early restart, transferencia en desorden
8) Buffers de escritura
9) Optimizaciones de código para reducir tasa fallos
10) Prefetch de instrucciones y/datos para reducir tasa de fallos y/o la penalización por fallo

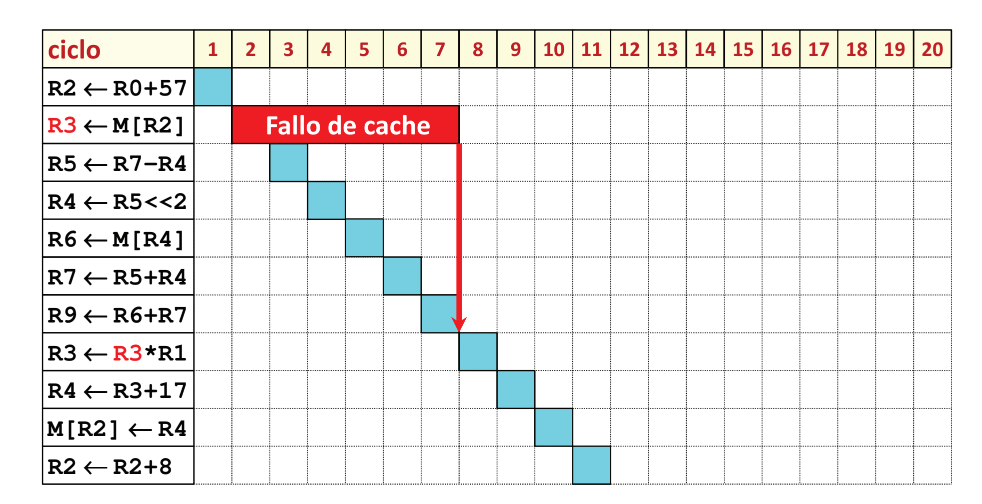
Non Blocking Cache

ciclo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
R2 ← R0+57	■																			
R3 ← M[R2]		Fallo de cache																		
R5 ← R7–R4								■												
R4 ← R5<<2									■											
R6 ← M[R4]										■										
R7 ← R5+R4											■									
R9 ← R6+R7												■								
R3 ← R3*R1													■							
R4 ← R3+17														■						
M[R2] ← R4															■					
R2 ← R2+8																■				

- Hasta ahora, cuando se produce un fallo de cache el procesador no inicia la ejecución de nuevas instrucciones hasta que se resuelve el fallo.
- En realidad, el procesador podría seguir ejecutando instrucciones mientras que no necesite R3.

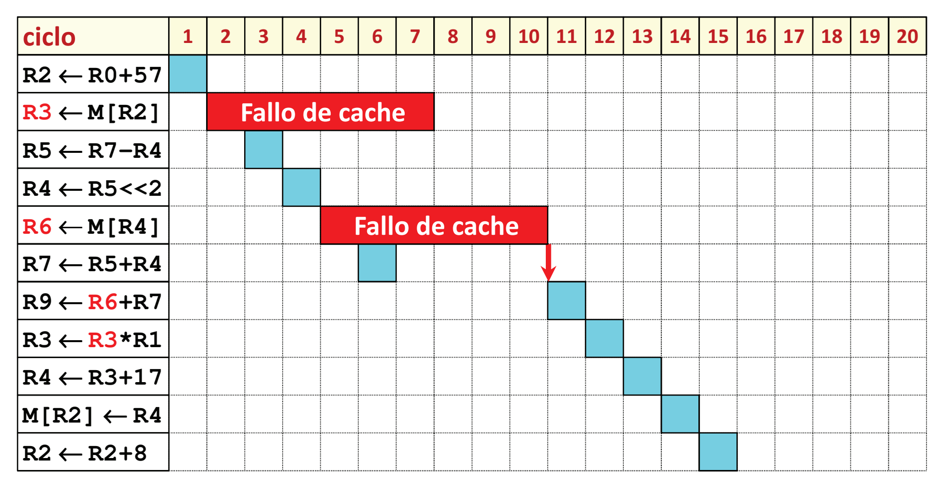


Non Blocking Cache



ciclo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
R2 ← R0+57	■																			
R3 ← M[R2]		Fallo de cache																		
R5 ← R7–R4			■																	
R4 ← R5<<2				■																
R6 ← M[R4]					■															
R7 ← R5+R4						■														
R9 ← R6+R7							■													
R3 ← R3*R1								■												
R4 ← R3+17									■											
M[R2] ← R4										■										
R2 ← R2+8											■									

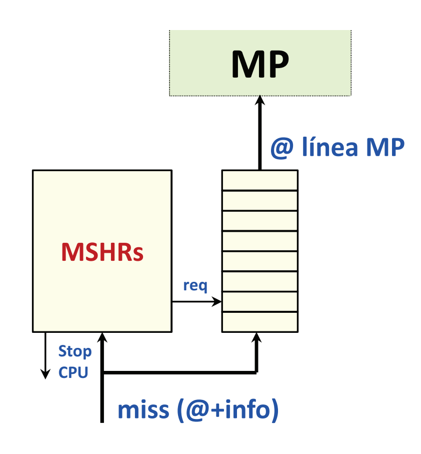
- En una Non Blocking Cache, cuando se produce un fallo de cache el procesador continúa la ejecución de instrucciones y sólo se detiene cuando necesita el dato que ha provocado el fallo de cache.

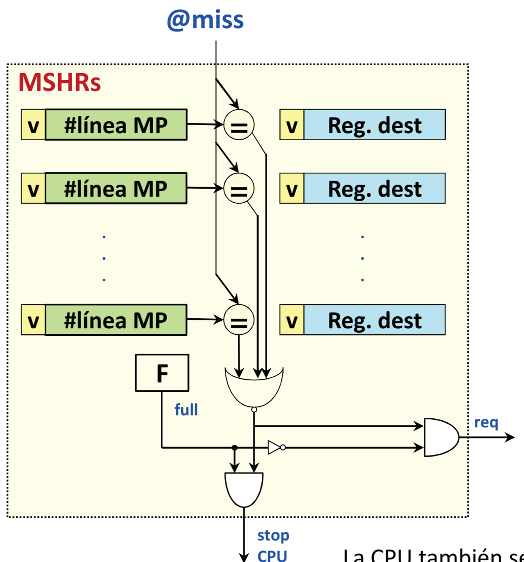


ciclo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
R2 ← R0+57																				
R3 ← M[R2]		Fallo de cache																		
R5 ← R7−R4																				
R4 ← R5<<2																				
R6 ← M[R4]				Fallo de cache																
R7 ← R5+R4																				
R9 ← R6+R7																				
R3 ← R3*R1																				
R4 ← R3+17																				
M[R2] ← R4																				
R2 ← R2+8																				

- Una Non Blocking Cache usa MSHRs (Miss Status Handler Register) para gestionar los fallos pendientes.
- El número de MSHRs condiciona el número de fallos que puede soportar la MC sin detener el procesador.
- La Idea original de los MSHRs es que el compilador/programador separe lo suficiente los accesos a memoria (**R3 ← M[R2]**) de su uso (**R3 ← R3*R1**) para que en caso de fallo de cache el procesador no se detenga.
- Los MSHRs mantienen información de cual es el registro destino del load para controlar cuando se ha de bloquear el procesador.



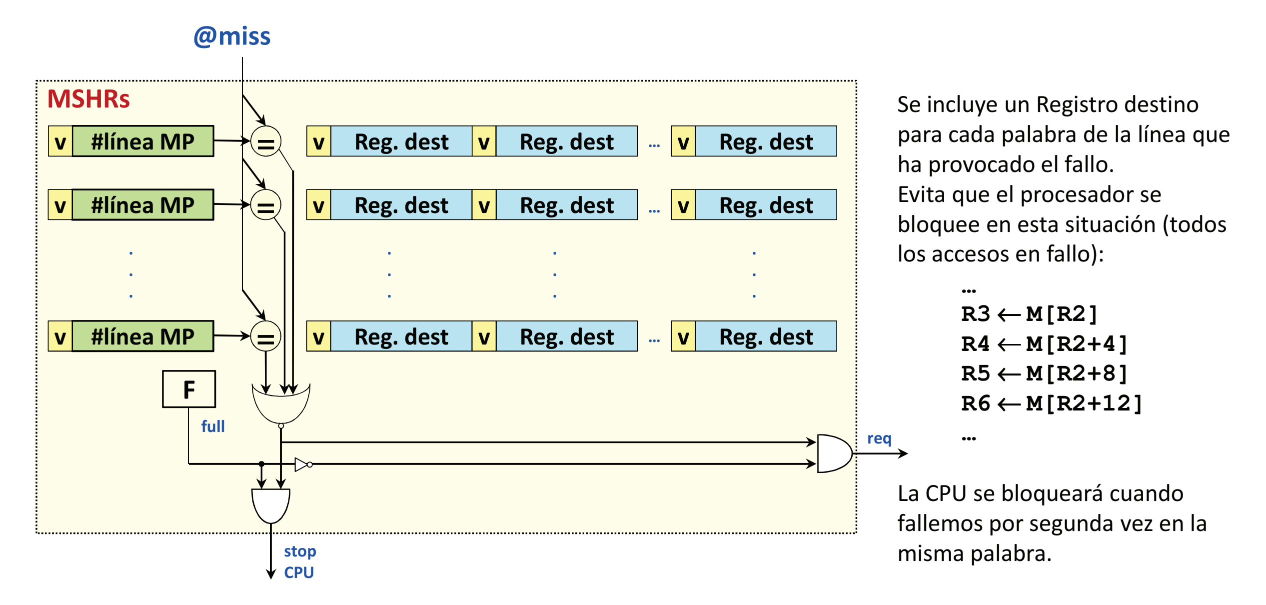




La CPU también se bloquea cuando se produce un segundo fallo sobre la misma línea, aunque sea en palabras diferentes. [Hardware no incluido en la figura]

David Kroft. «Lockup-free instruction fetch/prefetch cache organization», Proceedings of the 8th annual symposium on Computer Architecture (ISCA'81).

Non Blocking Cache



Se incluye un Registro destino para cada palabra de la línea que ha provocado el fallo.
Evita que el procesador se bloquee en esta situación (todos los accesos en fallo):

```
…
R3 ← M[R2]
R4 ← M[R2+4]
R5 ← M[R2+8]
R6 ← M[R2+12]
…
```

La CPU se bloqueará cuando fallemos por segunda vez en la misma palabra.

Las Non Blocking caches son imprescindibles en las CPUs con ejecución fuera de orden.

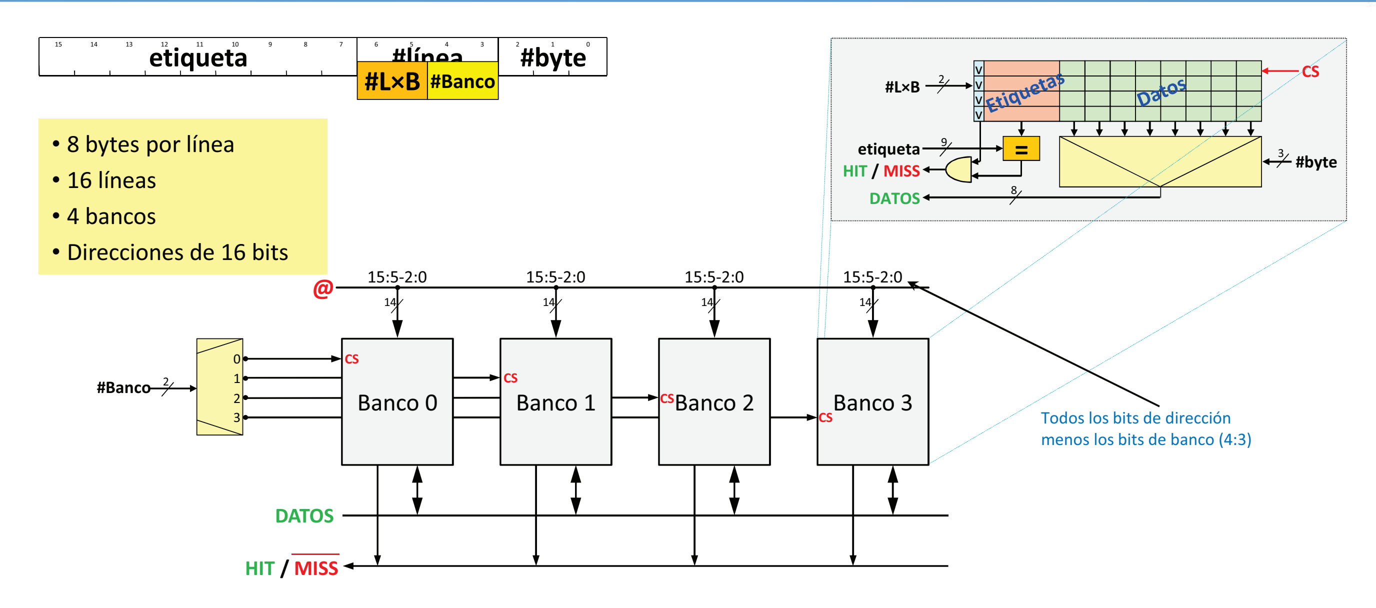
K. I. Farkas and N. P. Jouppi . «Complexity/performance tradeoffs with non-blocking loads», Proceedings of the 21th annual symposium on Computer Architecture (ISCA'94).



Optimizaciones

1) Caches pequeñas y simples para reducir el tiempo de acceso en acierto
2) Predicción de vía para reducir el tiempo de acceso en acierto
3) Trace Caches para reducir el tiempo de acceso en acierto
4) Caches segmentadas para aumentar el ancho de banda de la cache
5) Non Blocking caches
6) **Caches multibanco**
7) Reducir la penalización por fallo: early restart, transferencia en desorden
8) Buffers de escritura
9) Optimizaciones de código para reducir tasa fallos
10) Prefetch de instrucciones y/datos para reducir tasa de fallos y/o la penalización por fallo

Cache organizada en Bancos

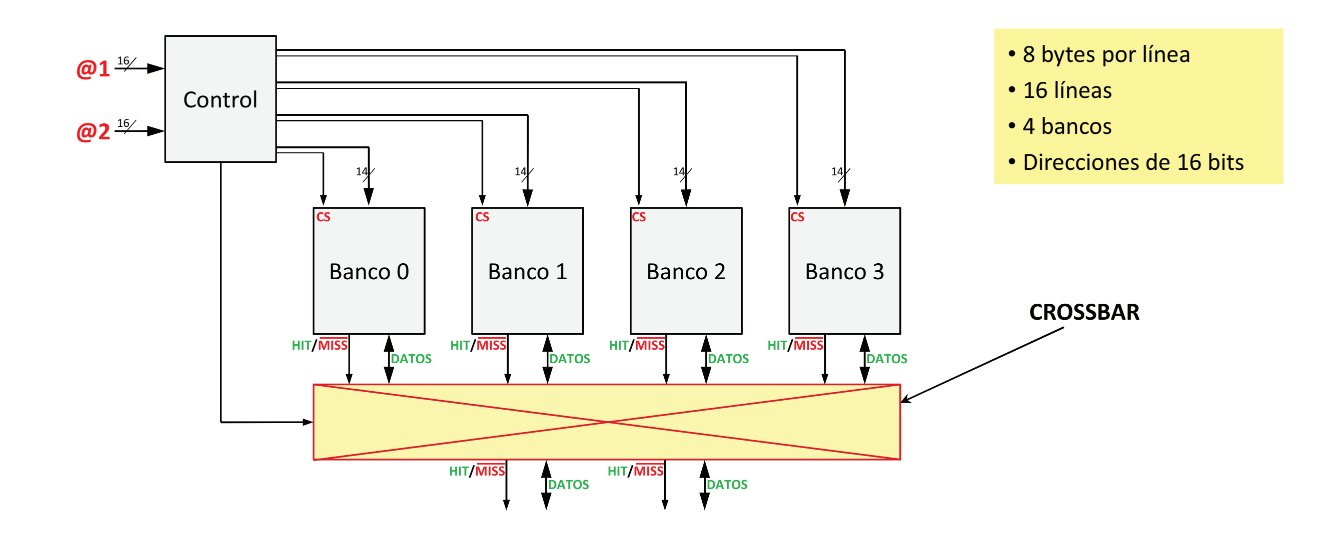


- **Esta organización permite:**
  - Reducir consumo, sólo es necesario activar el banco al que se accede.
  - Realizar accesos concurrentes.



Cache organizada en Bancos

- **2 Accesos simultáneos**



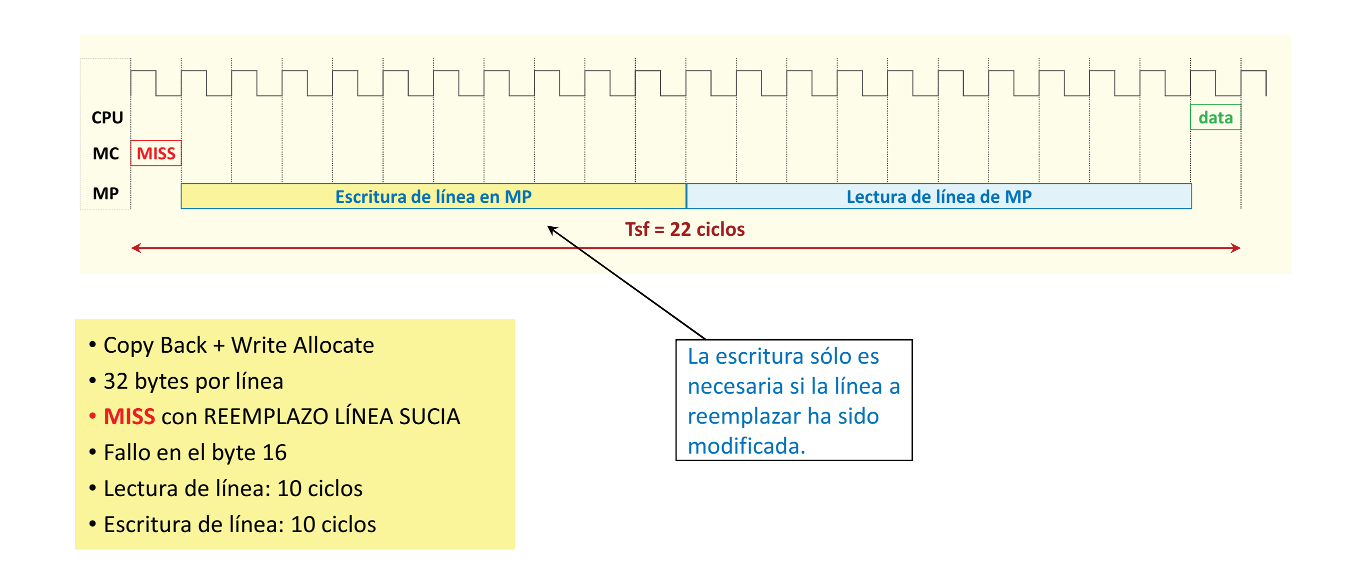
Optimizaciones

1) Caches pequeñas y simples para reducir el tiempo de acceso en acierto
2) Predicción de vía para reducir el tiempo de acceso en acierto
3) Trace Caches para reducir el tiempo de acceso en acierto
4) Caches segmentadas para aumentar el ancho de banda de la cache
5) Non Blocking caches
6) Caches multibanco
7) **Reducir la penalización por fallo: early restart, transferencia en desorden**
8) Buffers de escritura
9) Optimizaciones de código para reducir tasa fallos
10) Prefetch de instrucciones y/datos para reducir tasa de fallos y/o la penalización por fallo



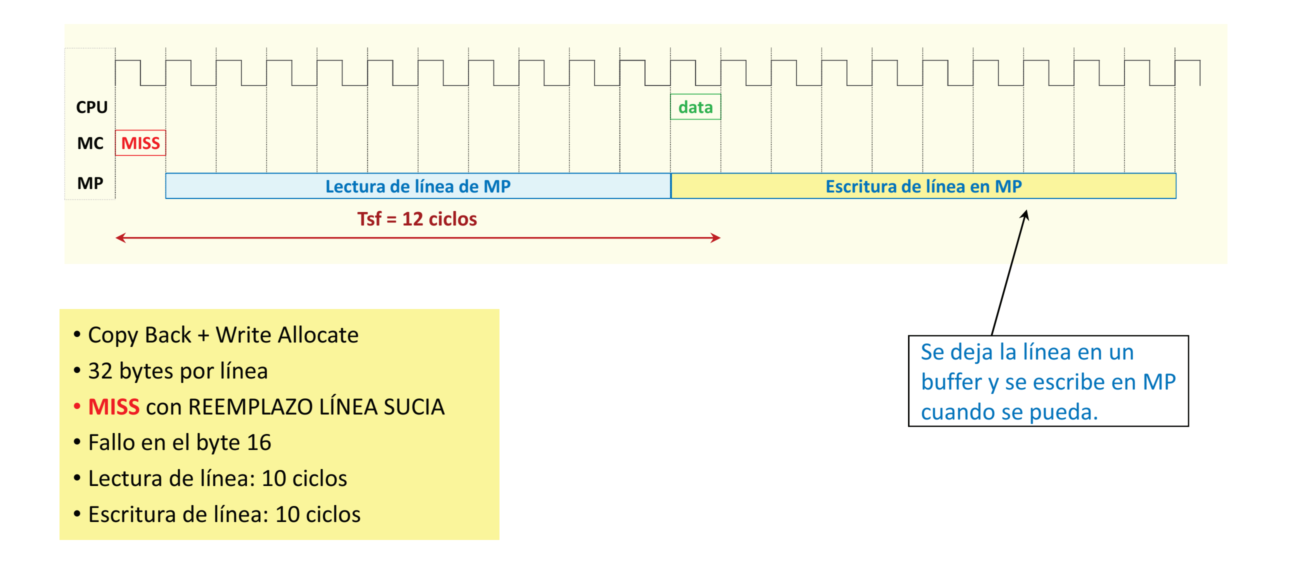
Reducir la penalización por fallo

- **Punto de partida**



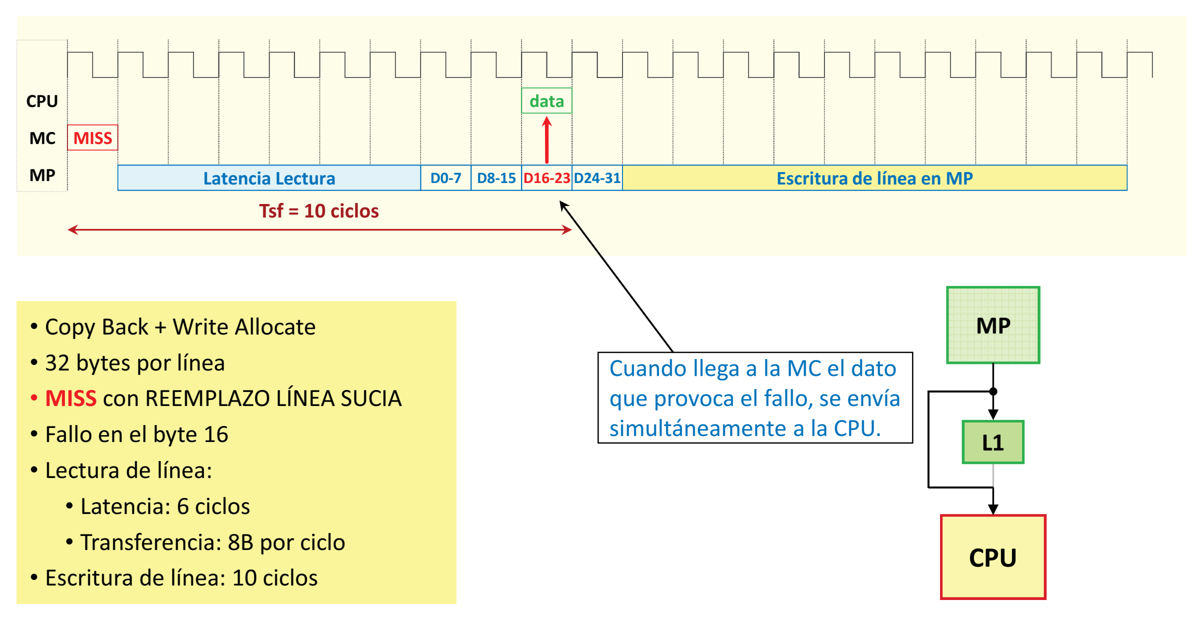
Reducir la penalización por fallo

- **Actualizar MP después de leer línea**



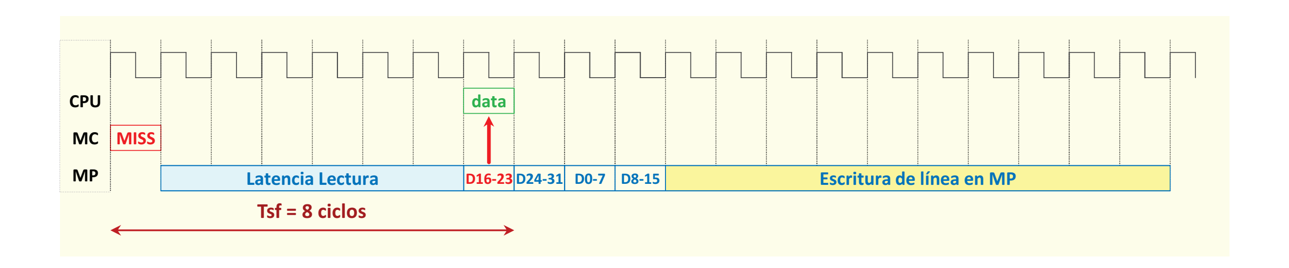
Reducir la penalización por fallo

- **Continuación Anticipada (*Early Restart*): en cuanto llega el dato que provoca el fallo, se envía al procesador**



Reducir la penalización por fallo

- **Continuación Anticipada (*Early Restart*): en cuanto llega el dato que provoca el fallo, se envía al procesador**



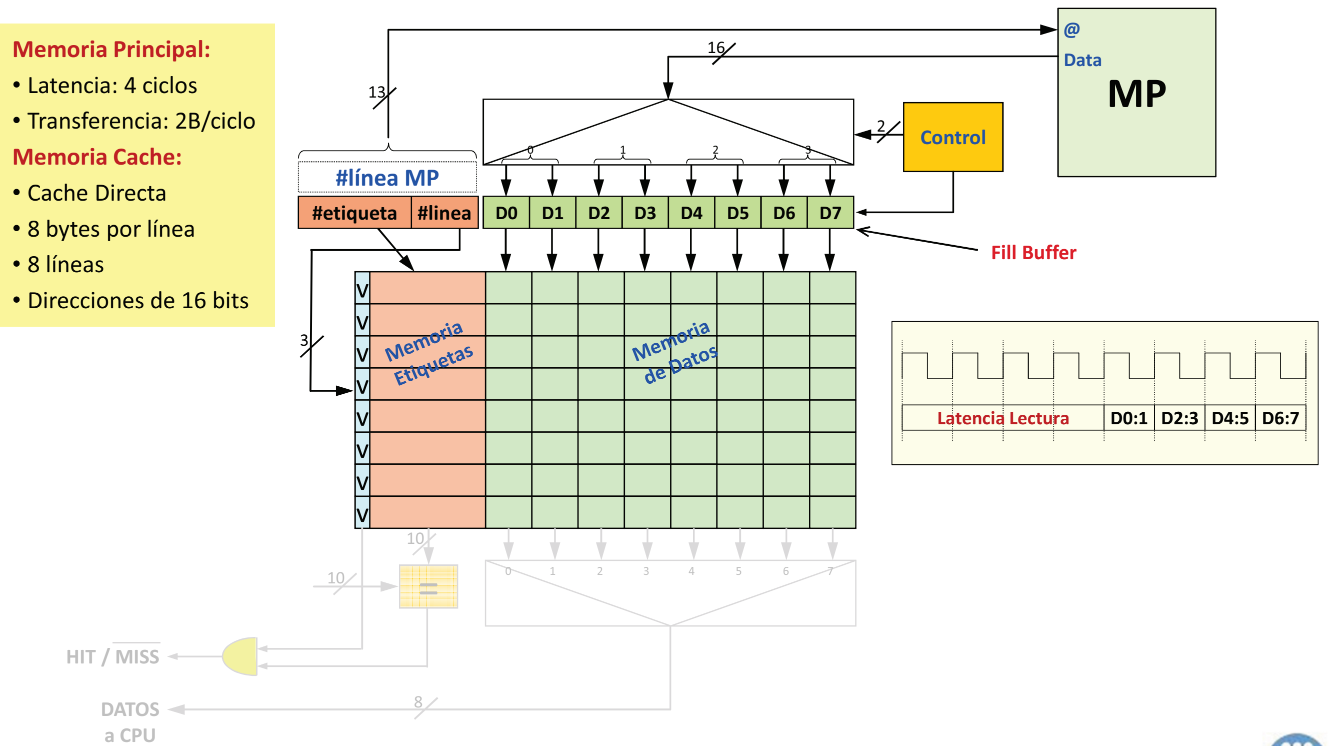
- Copy Back + Write Allocate
- 32 bytes por línea
- MISS con REEMPLAZO LÍNEA SUCIA
- Fallo en el byte 16
- Lectura de línea:
  - Latencia: 6 ciclos
  - Transferencia: 8B por ciclo
- Escritura de línea: 10 ciclos

En todos los casos, y para que estos mecanismos sean efectivos, cuando se pasa el dato al procesador y mientras se acaba de servir el fallo, la MC ha de ser capaz de recibir nuevos accesos (Cache no bloqueante).

Lectura Línea de MP

Memoria Principal:
- Latencia: 4 ciclos
- Transferencia: 2B/ciclo

Memoria Cache:
- Cache Directa
- 8 bytes por línea
- 8 líneas
- Direcciones de 16 bits



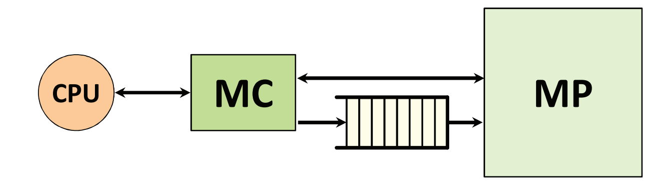
Optimizaciones

1) Caches pequeñas y simples para reducir el tiempo de acceso en acierto
2) Predicción de vía para reducir el tiempo de acceso en acierto
3) Trace Caches para reducir el tiempo de acceso en acierto
4) Caches segmentadas para aumentar el ancho de banda de la cache
5) Non Blocking caches
6) Caches multibanco
7) Reducir la penalización por fallo: early restart, transferencia en desorden
8) **Buffers de escritura**
9) Optimizaciones de código para reducir tasa fallos
10) Prefetch de instrucciones y/datos para reducir tasa de fallos y/o la penalización por fallo



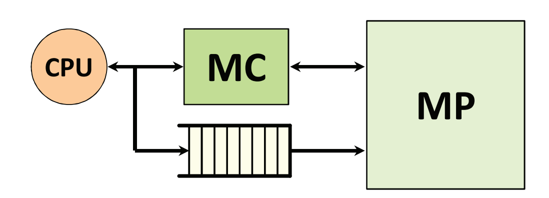
Buffers de Escritura

- **Si la cache es COPY BACK para reducir la penalización en caso de fallo hay que dar prioridad a leer la línea que contiene el dato que provoca el fallo a la escritura en MP de la línea reemplazada.**
- **Solución: poner un buffer de n entradas entre la MC y MP.**
  - Cuando se reemplaza una línea sucia se deja en el buffer
  - El acceso al buffer es rápido (equivale a acceder a MC, 1 ciclo).
  - Las escrituras se realizan cuando el bus entre MC y MP no está ocupado.
  - Unas pocas entradas son suficientes.
  - Cuando se realiza un acceso a MC, también hay que consultar en el buffer las líneas pendientes de escribir



Buffers de Escritura

- **Si la cache es WRITE THROUGH el coste de una escritura es el coste de escribir en Memoria Principal (no es aceptable).**
- **Solución: poner un buffer de n entradas entre la CPU y MP.**
  - Las escrituras se almacenan en el buffer
  - El acceso al buffer es rápido (equivale a acceder a MC, 1 ciclo).
  - Las escrituras se realizan cuando el bus entre MC y MP no está ocupado.
  - Unas pocas entradas son suficientes.
  - Cuando se realiza un acceso a MC, también hay que consultar en el buffer las datos pendientes de escribir



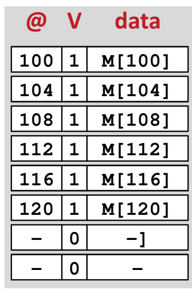


Buffers de Escritura

- **Situación común**

```
for (i=0; i<N; i++)
  V[i] = A[i] + B[i];
```

- **Cuando realizamos escrituras con localidad espacial, el buffer se llenará muy rápido y será poco eficiente.**
- **Además, las escrituras van a parar a la misma línea de cache.**
- **Solución: Merge Buffers**



@	V	data
100	1	M[100]
104	1	M[104]
108	1	M[108]
112	1	M[112]
116	1	M[116]
120	1	M[120]
–	0	–]
–	0	–

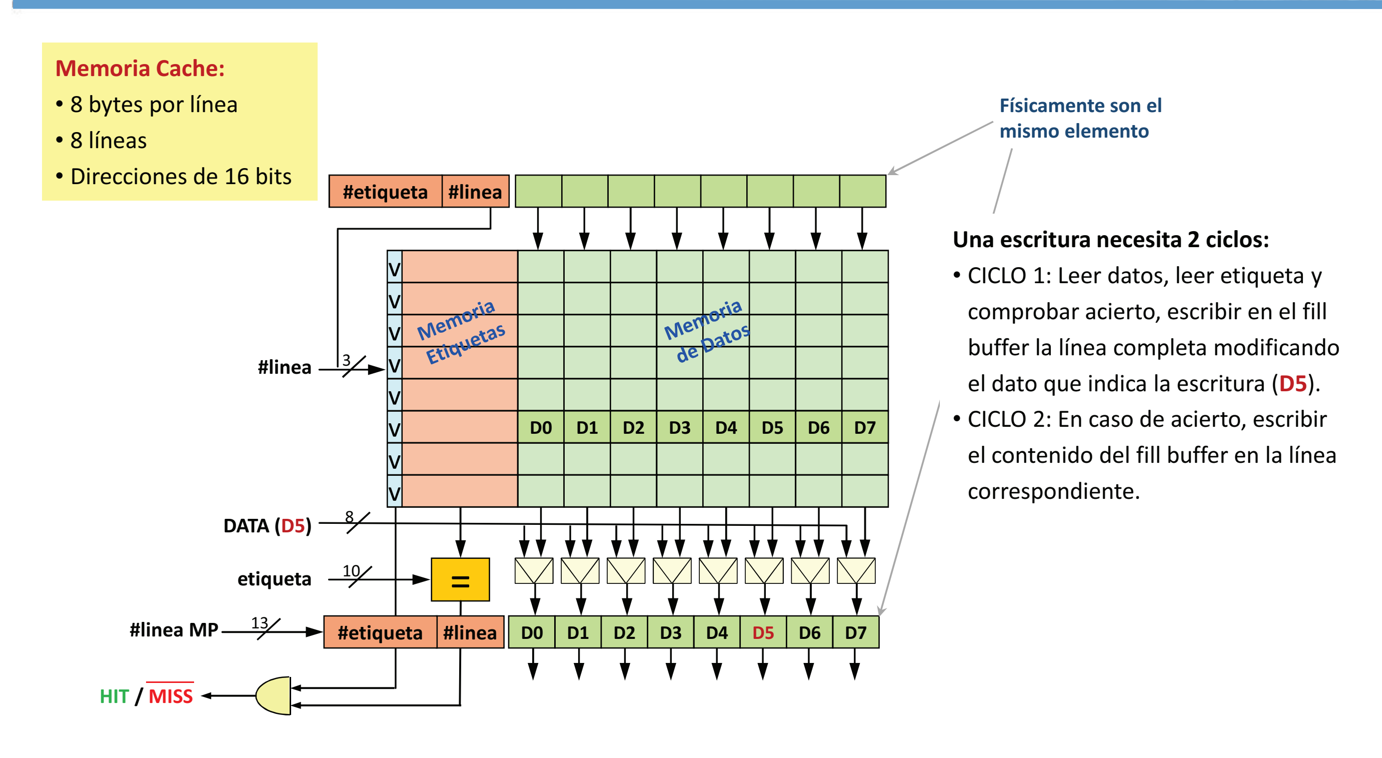
Buffer convencional

@	V	data	V	data	V	data	V	data
100	1	M[100]	1	M[104]	1	M[108]	1	M[112]
116	1	M[116]	1	M[120]	0	–	0	–
–	0	–	0	–	0	–	0	–

Merge Buffer

Se puede aprovechar el hecho de que escribir un bloque contiguo de memoria tiene prácticamente el mismo coste que escribir una palabra.

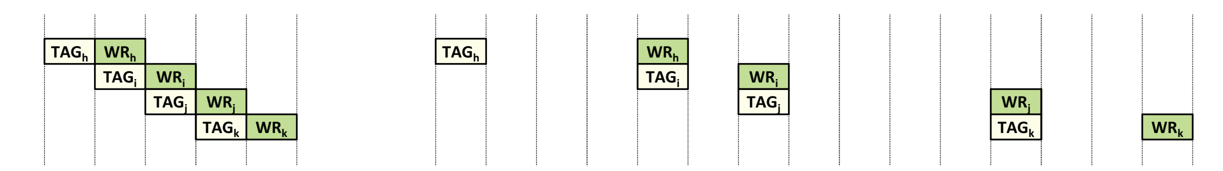
Escritura dato en MC





Reducir el coste de las escrituras

- **Cuando se realiza una lectura, se puede leer el dato en paralelo con la comprobación de si es un acierto o un fallo.**
- **Una escritura en MC no se puede realizar hasta que sabemos que es un acierto.**
- **Si una lectura nos cuesta 1 ciclo ⇒ una escritura cuesta 2 ciclos.**
- **Solución: ESCRITURAS SEGMENTADAS**
  - En el primer ciclo se comprueba si es acierto o fallo y en caso de acierto se deja el dato a escribir en un registro intermedio.
  - En la siguiente escritura (mientras se comprueba si es acierto o fallo) se realiza la escritura anterior.
  - Una escritura individual sigue tardando 2 ciclos, pero desde el punto de vista del procesador sólo cuesta un ciclo.



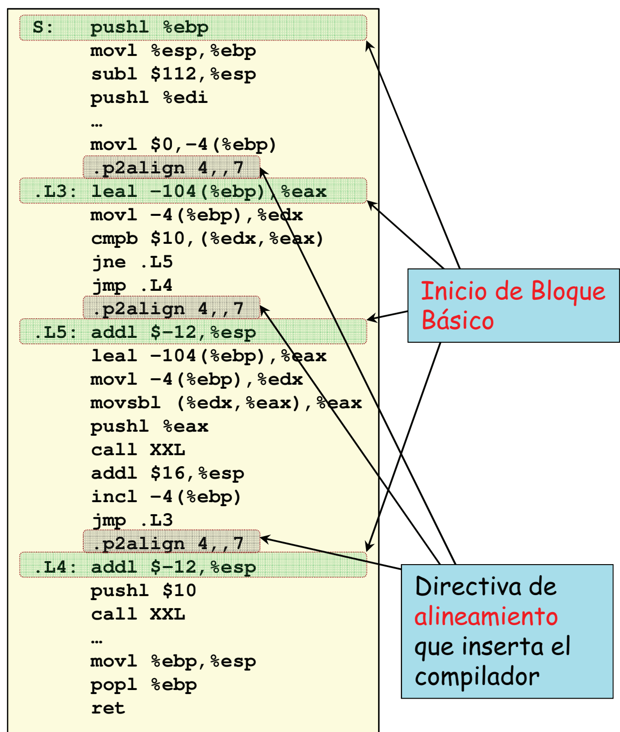
Optimizaciones

1) Caches pequeñas y simples para reducir el tiempo de acceso en acierto
2) Predicción de vía para reducir el tiempo de acceso en acierto
3) Trace Caches para reducir el tiempo de acceso en acierto
4) Caches segmentadas para aumentar el ancho de banda de la cache
5) Non Blocking caches
6) Caches multibanco
7) Reducir la penalización por fallo: early restart, transferencia en desorden
8) Buffers de escritura
9) **Optimizaciones de código para reducir tasa fallos**
10) Prefetch de instrucciones y/datos para reducir tasa de fallos y/o la penalización por fallo



Optimizaciones de código para reducir tasa fallos

- **Reordenación de código**
  - Alineando los puntos de entrada de los bloques básicos con el inicio de la línea de cache aumenta la probabilidad de acierto en cache para código secuencial.
  - Si el compilador intuye que un salto condicional se comportará la mayoría de las veces en el mismo sentido, puede organizar el código para que en ese caso el código se ejecute en secuencia.



Optimizaciones de código para reducir tasa fallos

Reordenación de los datos

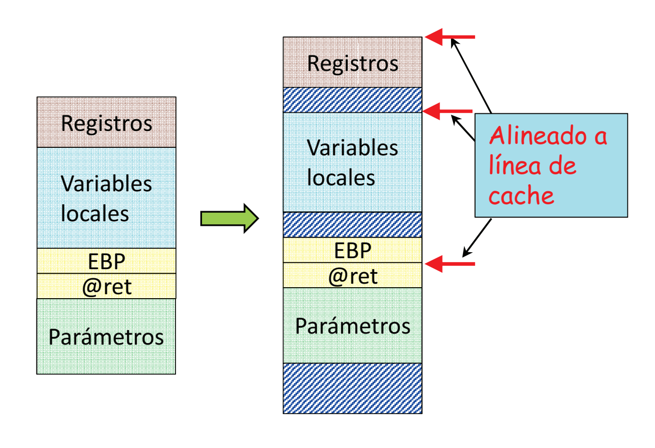
- Los datos pueden ubicarse en memoria para evitar conflictos.

```
// Código Original
int v[1024],w[1024];
int A[1024][1024];
```

```
// Código Transformado
int v[1024],d[p],w[1024];
int A[1024][1024+k];
```

El objetivo es evitar que las direcciones iniciales de los vectores y las filas de la matriz se mapeen en el mismo bloque (o conjunto) de cache.

- Las diferentes partes del bloque de activación de una subrutina pueden alinearse con el inicio de la línea de cache para aprovechar la localidad espacial.



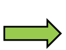


Optimizaciones de código para reducir tasa fallos

Loop Fussion (Fusión de bucles)

```
// Código Original
for(i=0; i < N; i++)
  a[i] = b[i] * c[i];

for(i=0; i < N; i++)
  d[i] = a[i] * c[i];
```



```
// Código Transformado
for(i=0; i < N; i++){
  a[i] = b[i] * c[i];
  d[i] = a[i] * c[i];
}
```

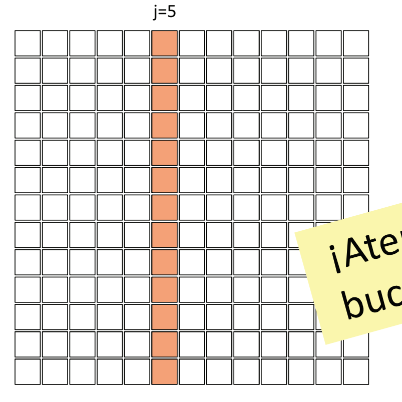
¿Qué tipo de localidad estamos aprovechando?

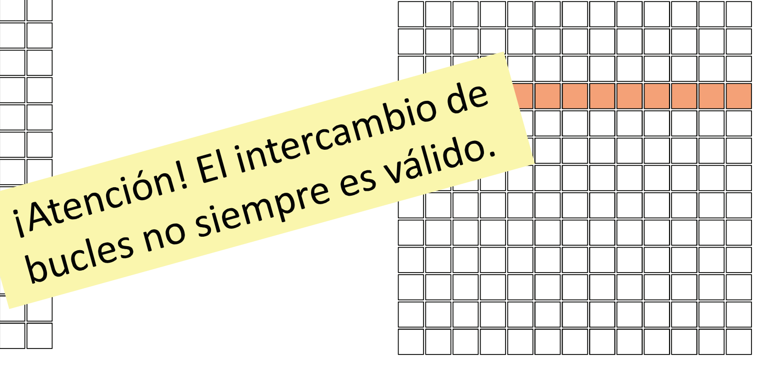
Optimizaciones de código para reducir tasa fallos

Loop Interchange (Intercambio de bucles)

```
// Código Original
for (j=0; j<100; j++)
  for (i=0; i<100; i++)
    x[i][j] = 2 * x[i][j];
```

```
// Código Transformado
for (i=0; i<100; i++)
  for (j=0; j<100; j++)
    x[i][j] = 2 * x[i][j];
```





Los accesos consecutivos a memoria de datos están separados por 100·4 bytes.

Los accesos consecutivos a memoria de datos están separados por 4 bytes. ¡Se aprovecha la localidad espacial!

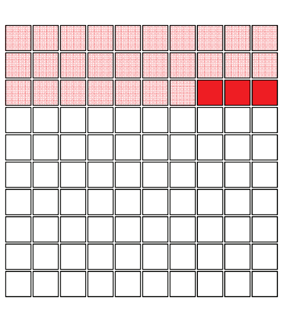


Optimizaciones de código para reducir tasa fallos

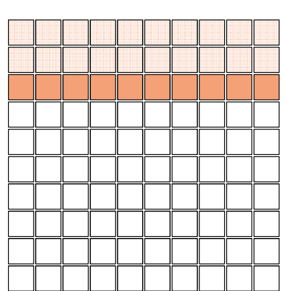
Blocking

```
// Código Original
for (i=0; i<N; i++)
  for (j=0; j<N; j++) {
    r=0;
    for (k=0; k<N; k++)
      r=r+y[i][k]*z[k][j];
    x[i][j]=r;
  }
```

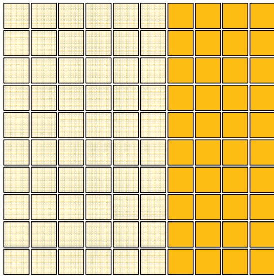
- Hay accesos a fila y columna, el intercambio de bucles no es suficiente.
- Para una i determinada se accede a y[i][*] N veces y se recorre completamente la matriz x.
- Suponiendo que no hay fallos por conflicto necesitamos que en la cache quepan N^2+N+1 elementos para que no se produzcan fallos de capacidad.
- Si la cache es menor se producirán fallos de capacidad. Como máximo serían $2N^3+N^2$ accesos a memoria para $2N^3$ operaciones.



=



·



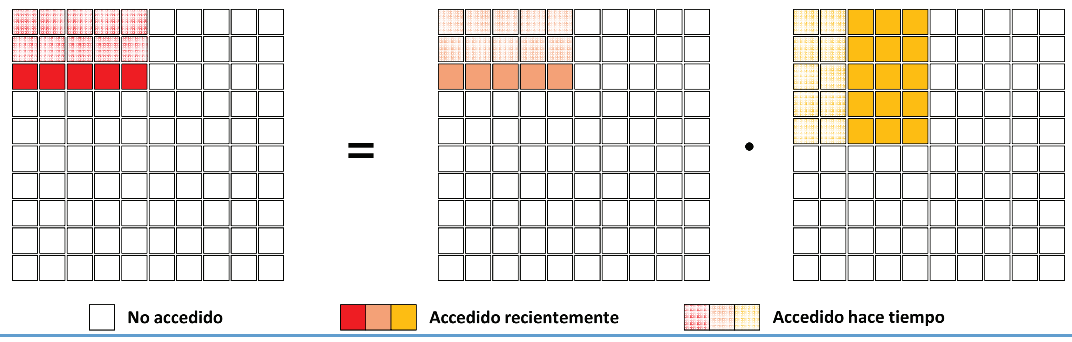
No accedido  Accedido recientemente Accedido hace tiempo

Optimizaciones de código para reducir tasa fallos

Blocking

```
// Código Transformado
for (jj=0; jj<N; jj=jj+B)
for (kk=0; kk<N; kk=kk+B)
for (i=0; i<N; i++)
  for (j=jj; j<min(jj+B,N); j++) {
    r=0;
    for (k=kk; k<min(kk+B,N); k++)
      r=r+y[i][k]*z[k][j];
    x[i][j]=r;
  }
```

- El objetivo del blocking es que los datos a los que se accede en los bucles más internos quepan en la cache.
- El parámetro B (*blocking factor*) se ajusta al tamaño de la cache disponible.
- Suponiendo que no hay fallos por conflicto necesitamos que en la cache quepan $N^2/B+N/B+1$ elementos para que no se produzcan fallos de capacidad.



No accedido — Accedido recientemente — Accedido hace tiempo



Optimizaciones de código para reducir tasa fallos

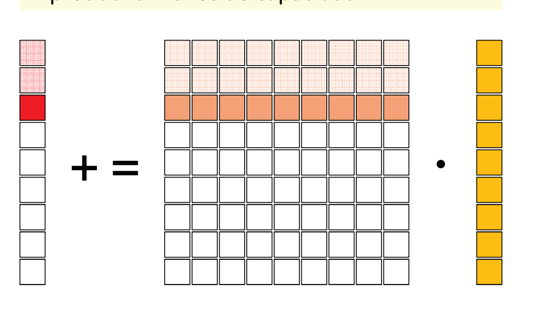
Blocking

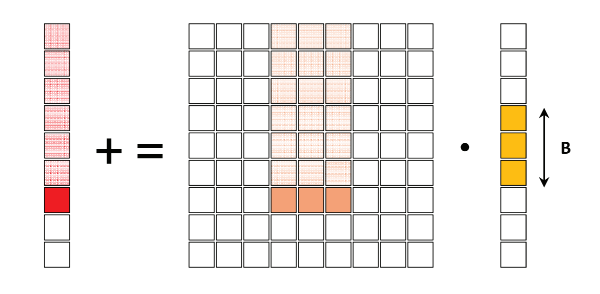
```
// Código Original
int A[N][N],x[N],y[N];
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    x[i] += A[i][j]*y[j]
```

```
// Código Transformado
for (jj=0; jj<N; jj=jj+B)
  for (i=0; i<N; i++)
    for (j=jj; j<min(N,jj+B); j++)
      x[i] += A[i][j]*y[j]
```

- La matriz se recorre por filas, se aprovecha la localidad espacial.
- Si el vector y es más grande que la cache se producirán fallos de capacidad.

- B se escoge de tal forma que la porción del vector y que se utiliza en cada iteración de jj quepa en la MC: aprovechamos la localidad temporal.





No accedido — Accedido recientemente — Accedido hace tiempo

Optimizaciones de código para reducir tasa fallos

- **La Memoria Cache es transparente al programador. Sin embargo, tenerla en cuenta en la programación puede mejorar (a veces espectacularmente) el rendimiento de los programas.**
- **Existen numerosas técnicas de programación / compilación para optimizar el rendimiento de la jerarquía de memoria.**
- **Las optimizaciones se hacen pensando en todos los niveles de la jerarquía:**
  - Registros del procesador
  - Memoria Cache
  - TLB




Optimizaciones

1) Caches pequeñas y simples para reducir el tiempo de acceso en acierto
2) Predicción de vía para reducir el tiempo de acceso en acierto
3) Trace Caches para reducir el tiempo de acceso en acierto
4) Caches segmentadas para aumentar el ancho de banda de la cache
5) Non Blocking caches
6) Caches multibanco
7) Reducir la penalización por fallo: early restart, transferencia en desorden
8) Buffers de escritura
9) Optimizaciones de código para reducir tasa fallos
10) **Prefetch de instrucciones y/datos para reducir tasa de fallos y/o la penalización por fallo**

Prefetch

- **Objetivo: Reducir los fallos de CARGA**
- **Estrategia: Especulamos con la localidad y traemos a MC aquella información que creemos que será utilizada en un futuro cercano, antes de que sea solicitada.**
  - Los accesos a instrucciones son más sencillos de predecir que los accesos a datos.
  - La información ha de llegar a tiempo, ni muy pronto ni demasiado tarde.
- **El problema del prefetch es que podemos traer información NO ÚTIL a la cache, ocupando espacio de MC y ancho de banda entre MC y MP.**
- **Tipos de prefetch:**
  - Prefetch hardware (datos e instrucciones)
  - Prefetch software (datos)

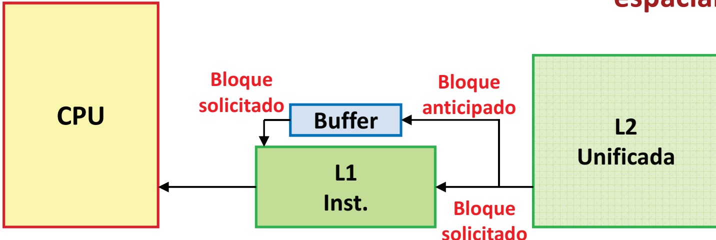


Prefetch Hardware

Prefetch Hardware de instrucciones en el Alpha 21064

- Cuando se produce un fallo de cache, se trae el bloque solicitado (**bloque i**) y el siguiente en secuencia (**bloque i+1**), si no está ya en la cache.
- El bloque solicitado se deja en la MC y el siguiente en el buffer de prefetch.
- Si hay fallo en cache, pero acierto en el buffer, se sirve el fallo desde el buffer, se pasa el bloque i+1 a la MC y se trae al buffer el siguiente (**bloque i+2**)
- El buffer puede tener varias entradas

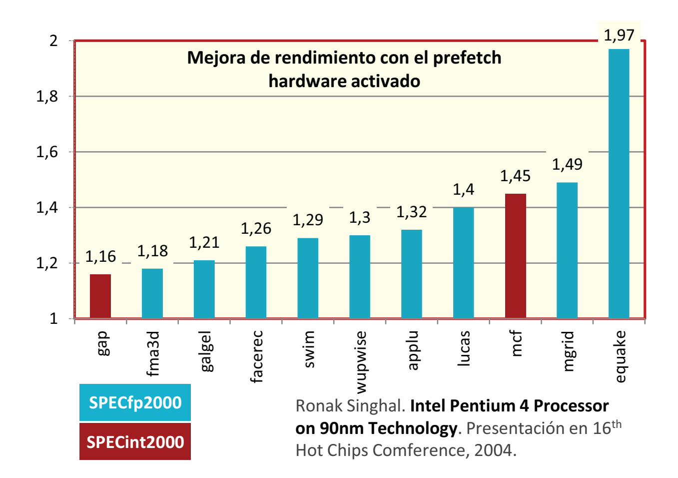
Esquema útil cuando hay **localidad espacial** (instrucciones).



Prefetch Hardware

Prefetch Hardware de datos

- Prefetch en fallo
  - Prefetch del bloque i+1 cuando se falla en el bloque i
- Esquema OBL (One Block Lookahead)
  - Prefetch del bloque i+1 cuando se accede al bloque i
- Prefetch con stride
  - Si se observa una secuencia de accesos a bloque **B**, **B+N**, **B+2·N**, entonces se hace prefetch de **B+3·N** …



Ronak Singhal. **Intel Pentium 4 Processor on 90nm Technology**. Presentación en 16th Hot Chips Comference, 2004.

- SPEC CPU2000
- Sólo se muestran los programas «más sensibles al prefetch».
- El prefetch hardware es especialmente eficaz con los programas de cálculo (SPECfp2000)



Prefetch Software (datos)

- Se utilizan instrucciones especiales (las insertan el compilador o bien el programador de LM) para traer los datos de forma anticipada.
- En general, el prefetch puede introducir tráfico (entre MP y MC) innecesario. Con el prefetch software tenemos más control (se puede reducir el tráfico inútil).
- Se pierde tiempo ejecutando instrucciones de prefetch (¿inútiles?).
- Ejemplo:

```
// Código Original
for(i=0; i < N; i++)
  sum = sum + b[i]*c[i];
```

⇒

```
// Código Transformado
for(i=0; i < N; i++){
  prefetch(&b[i+P]);
  prefetch(&c[i+P]);
  sum = sum + b[i]*c[i];
}
```

- El valor de P no es trivial de calcular:
  - Si hacemos prefetch muy cercano (P↓), es posible que el dato no llegue a tiempo.
  - Si hacemos prefetch muy lejano (P↑), tendremos polución en la cache.

Prefetch Software (datos)

- Normalmente, no ejecutaremos el prefetch en todas las iteraciones, lo haremos p.e. 1 de cada 4 (dependiendo del tamaño de línea de cache).
- Si suponemos que los vectores están alineados y que caben 4 elementos por línea, la transformación podría ser:

```
// Código Original
for(i=0; i < N; i++)
  sum = sum + b[i]*c[i];
```

→

```
// Código Transformado
for(i=0; i < N; i=i+4){
  prefetch(&b[i+4*P]);
  prefetch(&c[i+4*P]);
  sum = sum + b[i]*c[i];
  sum = sum + b[i+1]*c[i+1];
  sum = sum + b[i+2]*c[i+2];
  sum = sum + b[i+3]*c[i+3];
}
// P = 1, 2, 3, …
```



CPUs & Jerarquía (2005)

	AMD Opteron	Intel Pentium 4	IBM Power 5	Sun Niagara
ISA	80x86(64b)	80x86	PowerPC	SPARC v9
propósito	Sobremesa	Sobremesa	Servidor	Servidor
CMOS process (nm)	90	90	130	90
Die size (mm^2)	199	217	389	379
Inst issued/clock	3	3 RISC μops	8	1
cores	2	1	2	8
clock	2,8 GHz	3,6 GHz	2.0 GHz	1,2 GHz
SPECint2000	1924	1764	1510	-
SPECfp2000	1906	1994	3007	-
SPECweb2005	8669 (2 CPUs)	-	7881	14001

CPUs & Jerarquía (2005)

	AMD Opteron	Intel Pentium 4	IBM Power 5	Sun Niagara
L1 I por core	64 KB 2-way	trace cache (96KB)	64 KB 2-way	16 KB directa
Latencia L1 I (ciclos)	2	4	1	1
L1 D por core	64 KB 2-way	16 KB 8-way	32 KB 4-way	8 KB directa
Latencia L1 D (ciclos)	2	2	2	1
Entradas TLB (I/D/L2 I/ L2 D)	40/40/512/512	128/54	1024/1024	64/64
Min. Page size	4 KB	4 KB	4 KB	8 KB
On chip L2	2×1 MB 16-way	2 MB 8-way	1,875 MB 10-way	3 MB 2-way
Bancos L2	2	1	3	4
Latencia L2 (ciclos)	7	22	13	22 I, 23 D
Off chip L3 (tags on chip)	-	-	36 MB 12-way	
Latencia L3 (ciclos)	-	-	87	-
Tam Bloque (L1I/L1D/L2/L3) B	64	64/64/128/-	128/128/128/256	32/16/64/-
Bus Memoria (ancho bits)	128	64	64	128
Bus Memoria (clock)	200 MHz	200 MHz	400 MHz	400 MHz
Canales Memoria	1	1	4	4